Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 117 (2005) 263–284

www.elsevier.com/locate/entcs

New Evaluation Commands for Maude Within Full Maude[1]

Francisco Durán[a] Santiago Escobar[b] Salvador Lucas[b]

[a] *LCC, Universidad de Málaga, Campus de Teatinos, Málaga, Spain.* duran@lcc.uma.es

[b] *DSIC, UPV, Camino de Vera s/n, 46022 Valencia, Spain.* {sescobar,slucas}@dsic.upv.es

Abstract

Maude is able to deal with infinite data structures and avoid infinite computations by using *strategy annotations*. However, they can eventually make the computation of the normal form(s) of some input expressions impossible. We have used Full Maude to implement two new commands `norm` and `eval` which furnish Maude with the ability to compute (constructor) normal forms of initial expressions even when the use of strategy annotations together with the built-in computation strategy of Maude is not able to obtain them. These commands have been integrated into Full Maude, making them available inside the programming environment like any other of its commands.

Keywords: Declarative programming, Maude, reflection, strategy annotations.

1 Introduction

The ability of dealing with infinite objects is typical of lazy (functional) languages. Although the reductions in Maude are basically *innermost* (or eager), Maude is able to exhibit a similar behavior by using *strategy annotations* (see [17]). Maude strategy annotations are lists of non-negative integers associated to function symbols which specify the ordering in which the arguments are (eventually) evaluated in function calls: when considering a function call $f(t_1, \ldots, t_k)$, only the arguments whose indices are present as *positive* integers in the local strategy $(i_1 \cdots i_n)$ for f are evaluated (following the specified

[1] Work partially supported by CICYT TIC2001-2705-C03-01 and TIC2001-2705-C03-02, MCyT Acción Integrada HU 2003-0003, Agencia Valenciana de Ciencia y Tecnología GR03/025, and EU-India Cross-Cultural Dissemination project ALA/95/23/2003/077-054.

doi:10.1016/j.entcs.2004.06.014

ordering). If 0 is found, a reduction step on the whole term $f(t_1, \ldots, t_k)$ is attempted. The introduction of 'true' *replacement restrictions* (i.e., that forbid reductions on some arguments) is often sufficient to ensure (and even prove) a terminating behavior of a Maude program even though some expressions that denote an infinite object are involved (see [18] for an overview of methods to formally prove termination in these cases).

Full Maude is a language extension of Maude written in Maude itself, that endows Maude with notation for object-oriented programming and with a powerful and extensible module algebra in which Maude modules can be combined together to build more complex modules [9,10,4]. Every Maude module can be loaded in Full Maude by just enclosing it into parentheses. Then, the usual evaluation commands of Maude (e.g., `reduce`, `rewrite`, etc.) are available in Full Maude by also enclosing them into parentheses.

As a first example, let us consider the following parameterized module `LAZY-LIST` with a 'polymorphic' sort `List(X)`, and symbols `nil` (the empty list) and `_._` for the construction of polymorphic lists.

```
(fth TRIV is
   sort Elt .
 endfth)

(fmod LAZY-LIST(X :: TRIV) is
   protecting INT .
   sort List(X) .
   subsort X@Elt < List(X) .
   op nil : -> List(X) [ctor] .
   op _._ : X@Elt List(X) -> List(X) [ctor strat (1 0)] .
   op take : Int List(X) -> List(X) .
   var N : Int .   var X : X@Elt .   var Z : List(X) .
   eq take(0, Z) = nil .
   eq take(N, X . Z) = X . take(N - 1, Z) .
 endfm)
```

Note the strategy `(1 0)` associated to the operator `_._`, which forbids replacements on its second argument. Given a term of the form `X . L`, the strategy indicates that its first argument, the subterm `X`, will first be reduced, and then a reduction step on the whole term would be attempted. The `LAZY-LIST` module also includes a typical polymorphic operator `take` which selects the first n components of a list. Even though `take` has no explicit strategy annotation, Maude internally assigns a *by default* one `(1 2 0)`. In fact, Maude gives a strategy annotation $(1\ 2\ \cdots k\ 0)$ to each symbol f without an explicit strategy annotation.

The instantiation of the formal parameters of a parameterized module with actual parameter modules requires the use of *views*, which provide the interpretation of the actual parameters. Given a view `Nat` from the functional theory `TRIV` to the predefined module `NAT`, we may then define a function `natsFrom`, which is able to generate the infinite list of natural numbers, as

shown in the module NAT-LIST below.

```
(view Nat from TRIV to NAT is
   sort Elt to Nat .
 endv)

(fmod NAT-LIST is
   protecting LAZY-LIST(Nat) .
   op natsFrom : Nat -> List(Nat) .
   var N : Nat .
   eq natsFrom(N) = N . natsFrom(N + 1) .
 endfm)
```

Thanks to the strategy annotation (1 0) for the list constructor, after entering these modules into Full Maude, we can evaluate, e.g., the expression `take(4, natsFrom(0))` without entering into a non-terminating computation.

```
Maude> (red take(4, natsFrom(0)) .)
reduce in NAT-LIST : take(4, natsFrom(0))
result List‘(Nat‘) :
  0 . take(4 - 1, natsFrom(0 + 1))
```

This example shows a weak point of the use of strategy annotations: they may cause that the normal form(s) of input expressions become unreachable. In fact, from the user's point of view, this could be thought of as a kind of *incorrect* evaluation also, when normal forms are expected as the result of a computation. Removing the strategy annotations from module LAZY-LIST is not a solution (we would get a non-terminating program). On the other hand, there are different approaches to solve this problem and they can be summarized as follows:

(i) Performing a *layered normalization*: when the evaluation stops due to the replacement restrictions introduced by the strategy annotations, it is resumed over concrete inner parts of the resulting expression until the normal form is reached (if any) [15,16].

(ii) Transform the program to obtain a different one which is able to obtain sufficiently interesting outputs (e.g., constructor terms) [2].

(iii) Use *on-demand* strategy annotations where the presence of *negative* indices allows for some extra evaluation [1].

In this paper, we introduce new commands to make techniques (i) and (ii) available for the execution of Maude programs. Or to be more precise, of Full Maude programs. On-demand strategy annotations are not directly available in Maude, although they can be used, e.g., in CafeOBJ programs (see [22]).

Full Maude and its execution environment are implemented using the reflective capabilities of Maude. In fact, reflection, together with the good properties of rewriting logic as a logical framework [20,19], makes quite easy to develop formal tools and execution environments in Maude for any logic $\mathcal{L}$

of interest, including rewriting logic itself (see e.g. [10,3]). More attractive for our goals is the greater flexibility, maintainability, and extensibility that the *implementation* obtained in this way affords. In particular, in the same way commands like `reduce`, `search`, or `match` can be implemented in Full Maude, we could make new commands available. This is in fact what we have done, and what we present in this piece of work. Of course, instead of making direct calls to `metaReduce`, `metaSearch`, or `metaMatch`, as in the case of the above commands, we shall need to take the appropriate actions for each of them: to define a new rewriting strategy in the case of the `norm` command—technique (i)—or to provide the appropriate transformation in the case of the `eval` command—technique (ii).

The rest of the paper is structured as follows. Section 2 gives some preliminaries on context-sensitive rewriting and local strategies. Section 3 discusses on reflection and the way it appears in Maude through its built-in module `META-LEVEL`. Sections 4 and 5 explain, respectively, how the commands `norm` and `eval` have been added to Full Maude. Section 6 draws some conclusions and future work.

2 Preliminaries

Throughout the paper, $\mathcal{X}$ denotes a countable set of variables and $\mathcal{F}$ denotes a signature, i.e., a set of function symbols $\{f, g, \ldots\}$, each having a fixed arity given by a mapping $ar : \mathcal{F} \to \mathbb{N}$. The set of terms built from $\mathcal{F}$ and $\mathcal{X}$ is $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Terms are viewed as labelled trees in the usual way. Positions $p, q, \ldots$ are represented by chains of positive natural numbers used to address subterms of t. We denote the empty chain by Λ. Given positions p and q, we denote its concatenation as $p.q$. If p is a position, and Q is a set of positions, $p.Q = \{p.q \mid q \in Q\}$. The set of positions of a term t is $\mathcal{P}os(t)$. The subterm at position p of t is denoted as $t|_p$, and $t[s]_p$ is the term t with the subterm at position p replaced by s. The symbol labelling the root of t is denoted as $root(t)$.

A rewrite rule is an ordered pair (l, r), written $l \to r$, with $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$ and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. The left-hand side ($lhs$) of the rule is l and r is its right-hand side (rhs). A TRS is a pair $\mathcal{R} = (\mathcal{F}, R)$ where R is a set of rewrite rules. A term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ rewrites to s (at position p), written $t \xrightarrow{p}_{\mathcal{R}} s$ (or just $t \to s$), if $t|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$, for some rule $\rho : l \to r \in R$, $p \in \mathcal{P}os(t)$ and substitution σ. A TRS is terminating if $\to$ is terminating.

2.1 Context-sensitive rewriting

A mapping $\mu : \mathcal{F} \to \mathcal{P}(\mathbb{N})$ is a *replacement map* (or $\mathcal{F}$-map) if $\forall f \in \mathcal{F},\ \mu(f) \subseteq \{1, \ldots, ar(f)\}$ [12]. The set of *μ-replacing positions* $\mathcal{P}os^\mu(t)$ of $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is: $\mathcal{P}os^\mu(t) = \{\Lambda\}$, if $t \in \mathcal{X}$ and $\mathcal{P}os^\mu(t) = \{\Lambda\} \cup \bigcup_{i \in \mu(root(t))} i.\mathcal{P}os^\mu(t|_i)$, if $t \notin \mathcal{X}$. In *context-sensitive rewriting* (*CSR* [12]), we (only) contract *replacing* redexes: t μ-rewrites to s, written $t \hookrightarrow_\mu s$, if $t \xrightarrow{p}_{\mathcal{R}} s$ and $p \in \mathcal{P}os^\mu(t)$. The $\hookrightarrow_\mu$-normal forms are called μ-normal forms. Note that, except for the trivial case $\mu = \mu_\top$, where $\mu_\top(f) = \{1, \ldots, ar(f)\}$ for all $f \in \mathcal{F}$, the μ-normal forms strictly include all normal forms of $\mathcal{R}$. The canonical replacement map $\mu^{can}_{\mathcal{R}}$ is *the most restrictive replacement map ensuring that the non-variable subterms of the left-hand sides of the rules of $\mathcal{R}$ are replacing.* Note that $\mu^{can}_{\mathcal{R}}$ is easily obtained from $\mathcal{R}$: $\forall f \in \mathcal{F}, i \in \{1, \ldots, ar(f)\}$,

$$i \in \mu^{can}_{\mathcal{R}}(f) \quad \text{iff} \quad \exists l \in L(\mathcal{R}), p \in \mathcal{P}os_{\mathcal{F}}(l), (root(l|_p) = f \wedge p.i \in \mathcal{P}os_{\mathcal{F}}(l))$$

Let $CM_{\mathcal{R}} = \{\mu \in M_{\mathcal{R}} \mid \mu^{can}_{\mathcal{R}} \sqsubseteq \mu\}$ be the set of replacement maps which are less or equally restrictive than $\mu^{can}_{\mathcal{R}}$. One of the most important properties of the canonical replacement map(s) is the following.

Theorem 2.1 [12,15] *Let $\mathcal{R}$ be a left-linear TRS and $\mu \in CM_{\mathcal{R}}$. Every μ-normal form is a head-normal form.*

2.2 E-strategies

A *positive* local strategy (or E-strategy) for a k-ary symbol $f \in \mathcal{F}$ is a sequence $\varphi(f)$ of integers taken from $\{0, 1, \ldots, k\}$ which are given in parentheses (see the `LAZY-LIST` module in Section 1). A mapping φ that associates a local strategy $\varphi(f)$ to every $f \in \mathcal{F}$ is called an E-strategy map [22]. Nagaya describes the operational semantics of term rewriting under E-strategy maps as follows [21]: Let $\mathcal{L}$ be the set of all lists consisting of natural numbers. By $\mathcal{L}_n$, we denote the set of all lists of natural numbers not exceeding $n \in \mathbb{N}$. We use the signature $\mathcal{F}_{\mathcal{L}} = \{f_L \mid f \in \mathcal{F} \wedge L \in \mathcal{L}_{ar(f)}\}$ and labelled variables $\mathcal{X}_{\mathcal{L}} = \{x_{nil} \mid x \in \mathcal{X}\}$. An E-strategy map φ for $\mathcal{F}$ is extended to a mapping from $\mathcal{T}(\mathcal{F}, \mathcal{X})$ to $\mathcal{T}(\mathcal{F}_{\mathcal{L}}, \mathcal{X}_{\mathcal{L}})$ as follows:

$$\varphi(t) = \begin{cases} x_{nil} & \text{if } t = x \in \mathcal{X} \\ f_{\varphi(f)}(\varphi(t_1), \ldots, \varphi(t_k)) & \text{if } t = f(t_1, \ldots, t_k) \end{cases}$$

The mapping *erase* : $\mathcal{T}(\mathcal{F}_{\mathcal{L}}, \mathcal{X}_{\mathcal{L}}) \to \mathcal{T}(\mathcal{F}, \mathcal{X})$ removes labellings from symbols in the obvious way. The binary relation $\to_\varphi$ on $\mathcal{T}(\mathcal{F}_{\mathcal{L}}, \mathcal{X}_{\mathcal{L}}) \times \mathbb{N}^*_+$ (i.e., pairs $\langle t, p\rangle$ of labelled terms t and positions p) is [22,21]: $\langle t, p\rangle \to_\varphi \langle s, q\rangle$ if and only

if $p \in \mathcal{P}os(t)$ and either

(i) $root(t|_p) = f_{nil}$, $s = t$ and $p = q.i$ for some i; or

(ii) $t|_p = f_{i:L}(t_1, \ldots, t_k)$ with $i > 0$, $s = t[f_L(t_1, \ldots, t_k)]_p$ and $q = p.i$; or

(iii) $t|_p = f_{0:L}(t_1, \ldots, t_k)$, $erase(t|_p)$ is not a redex, $s = t[f_L(t_1, \ldots, t_k)]_p$, $q = p$; or

(iv) $t|_p = f_{0:L}(t_1, \ldots, t_k) = \sigma(l')$, $erase(l') = l$, $s = t[\sigma(\varphi(r))]_p$ for some $l \to r \in R$ and substitution σ, $q = p$.

We write $e \in L$ to denote that item e appears somewhere within the list L. Given an E-strategy map φ for $\mathcal{F}$, we define $\mu^\varphi \in M_\mathcal{F}$ as follows: for all $f \in \mathcal{F}$, $\mu^\varphi(f) = \{i \neq 0 \mid i \in \varphi(f)\}$. We will drop superscript φ if no confusion arises. Rewriting with strategy annotations is closely related to *CSR*.

Theorem 2.2 [13,14] *Let $\mathcal{R}$ be a TRS and φ be a positive E-strategy map. Let $t \in \mathcal{T}(\mathcal{F}_\mathcal{L}, \mathcal{X}_\mathcal{L})$, and $p \in \mathcal{P}os^\mu(erase(t))$ be s.t. $root(t|_p) = f_L$ for some suffix L of $\varphi(f)$. If $\langle t, p\rangle \to_\varphi \langle s, q\rangle$, then $q \in \mathcal{P}os^\mu(erase(s))$ and $erase(t) \hookrightarrow_\mu^= erase(s)$.*

This result expresses that any reduction sequence issued under control of an E-strategy φ consists of μ^φ-rewriting steps. Semantics of programs under a given E-strategy map φ is given by means of a mapping $eval_\varphi$ (from terms to their sets of 'computed values'). Nagaya describes $eval_\varphi$ for positive E-strategy maps by using the reduction relation $\to_\varphi$ [21,22]: given a TRS $\mathcal{R} = (\mathcal{F}, R)$ and a positive E-strategy map φ for $\mathcal{F}$, $eval_\varphi : \mathcal{T}(\mathcal{F}, \mathcal{X}) \to \mathcal{P}(\mathcal{T}(\mathcal{F}, \mathcal{X}))$ is defined as $eval_\varphi(t) = \{erase(s) \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \mid \langle \varphi(t), \Lambda\rangle \to_\varphi^! \langle s, \Lambda\rangle\}$. Terms collected in $eval_\varphi(t)$ are called E-normal forms. The following result shows that they are simply μ^φ-normal forms.

Theorem 2.3 [13] *Let $\mathcal{R} = (\mathcal{C} \uplus \mathcal{D}, R)$ be a TRS and φ be an E-strategy map such that for all $f \in \mathcal{D}$, $\varphi(f)$ ends in 0. If $s \in eval_\varphi(t)$, then s is a μ-normal form of t.*

Strategy annotations in Maude fulfill the requirement of Theorem 2.3. This is essential for the use of the first new evaluation command that we introduce in Section 4.

3 Reflection and the `META-LEVEL` module

Maude's design and implementation systematically exploits the reflective capabilities of rewriting logic [4], providing key features of the universal theory $\mathcal{U}$ in its built-in `META-LEVEL` module. In particular, `META-LEVEL` has sorts `Term` and `Module`, so that the representations of a term t and of a module $\mathcal{R}$ are, respectively, a term $\overline{t}$ of sort `Term` and a term $\overline{\mathcal{R}}$ of sort `Module`.

The basic cases in the representation of terms are obtained by subsorts `Constant` and `Variable` of the sort `Qid` of quoted identifiers. Constants are quoted identifiers that contain the name of the constant and its type separated by a dot, e.g., `'0.Nat`. Similarly, variables contain their name and type separated by a colon, e.g., `'N:Nat`. Then, a term is constructed in the usual way, by applying an operator symbol to a list of terms.

```
subsorts Constant Variable < Qid Term .
subsort Term < TermList .
op _,_ : TermList TermList -> TermList [ctor assoc] .
op _[_] : Qid TermList -> Term [ctor] .
```

For example, the term `take(0, 2 . nil)` of sort `List(Nat)` in the module `LAZY-LIST(Nat)` is metarepresented as

```
'take['0.Nat, '_._['2.Nat, 'nil.List`(Nat`)]]
```

Similarly, `META-LEVEL` includes declarations for metarepresenting modules. For example, a functional module can be represented as a term of sort `Module` using the following operator.

```
op fmod_is_sorts_.____endfm : Qid ImportList SortSet SubsortDeclSet
  OpDeclSet MembAxSet EquationSet -> FModule [ctor ...] .
```

Similar declarations allow us to represent the different types of declarations we can find in a module.

The module `META-LEVEL` also provides key metalevel functions for rewriting and evaluating terms at the metalevel, namely, `metaApply`, `metaRewrite`, `metaReduce`, etc., and also generic parsing and pretty printing functions `metaParse` and `metaPrettyPrint` [6,4]. For example, `metaReduce` takes as arguments the representation of a module $\mathcal{R}$ and the representation of a term t in that module, and returns the representation of the fully reduced form of the term t using the equations in $\mathcal{R}$, together with its corresponding sort or kind.

```
op metaReduce : Module Term -> [ResultPair] .
op {_,_} : Term Type -> ResultPair [ctor] .
```

All this functionality is very useful for metaprogramming, and in particular when building formal tools. Moreover, Full Maude provides a powerful setting in which additional facilities are available, making the implementation of new commands, as the ones introduced in this paper, much simpler. The specification of Full Maude and its execution environment can then be used as the infrastructure on which building new features, with the additional advantage of making such commands applicable to Full Maude modules, that is, to modules which may be parameterized, object-oriented, etc.

4 Extending Full Maude to handle the command `norm`

The *normalization via μ-normalization* procedure is introduced in [15,16] as a simple mechanism to furnish *CSR* with the ability to compute normal forms of initial expressions even though *CSR* itself is not able to compute them. The technique works for left-linear, confluent TRSs $\mathcal{R}$ which use a *canonical* replacement map, i.e., $\mu \in CM_{\mathcal{R}}$. The idea is very simple: if we are able to ensure that the μ-normal forms are always head-normal forms (and this is the case under the previous assumptions, see Theorem 2.1), then it is safe to get into the maximal non-replacing subterms of a μ-normal form s of a term t to (recursively) continue the computation.

We have implemented this procedure as a new command `norm` which uses the outcome of `red` (which, in Maude, are μ^{φ}-normal forms according to Theorem 2.3) to perform this layered evaluation of the initial expressions. The new command permits to obtain the intended value of the expression in Section 1.

```
Maude> (norm take(4, natsFrom(0)) .)
reduce in NAT-LIST :
  take(4, natsFrom(0))
result List‘(Int‘) :
  0 . 1 . 2 . 3 . nil
```

As for other commands, we may define the actions to take when the command is used by defining its corresponding meta-function. For instance, a `reduce` command is executed by appropriately calling the `metaReduce` function. In the case of `norm`, we define an operation `metaNorm`, which takes arguments of sort `Module` and `Term` and returns a term of sort `ResultPair`, a pair consisting of a term and its corresponding sort (see [6] for details). Basically, `metaNorm` calls the auxiliary function `metaNormRed`, which reduces the term using `metaReduce`, and then proceeds recursively on each of the arguments of the resulting term. It calls `metaNormRed` on the arguments with evaluation restrictions, and `metaNormNoRed` on the others. `metaNormNoRed` proceeds as `metaNormRed`, but without reducing the term before going on the arguments.

```
op metaNorm : Module Term -> ResultPair .
ops metaNormRed metaNormNoRed : Module Term -> Term .
op procStrat : Module Term OpDeclSet -> Term .
op procStrat : Module Term Nat NatList -> Term .

var  F : Qid .                 var  M : Module .
var  C : Constant .            var  OPDS : OpDeclSet .
var  V : Variable .            var  N : Nat .
vars T T' : Term .             var  NL : NatList .
var  TL : TermList .

ceq metaNorm(M, T)
  = {T', leastSort(M, T')}
  if T' := metaNormRed(M, T) .

eq metaNormRed(M, T)
  = procStrat(M, getTerm(metaReduce(M, T)), getOps(M)) .
```

```
eq metaNormNoRed(M, T) = procStrat(M, T, getOps(M)) .

eq procStrat(M, V, OPDS) = V .
eq procStrat(M, C, OPDS) = C .
eq procStrat(M, F[TL], OPDS)
  = F[procStratAux(M,TL,1,getStrat(M,F,getTypes(M,TL),OPDS))] .

eq procStratAux(M, (T, TL), N, NL)
  = procStratAux(M, T, N, NL), procStratAux(M, TL, N + 1, NL) .
eq procStratAux(M, T, N, NL)
  = if N in NL
    then metaNormNoRed(M, T)
    else metaNormRed(M, T)
    fi .
```

Now, we need to extend Full Maude so that the command can be entered by the user. To parse some input using the built-in function `metaParse`, we need to give the metarepresentation of the signature in which the input is going to be parsed. In Full Maude, such a grammar is provided by the `FULL-MAUDE-SIGN` module, in which we can find the appropriate declarations so that any valid input, namely modules, theories, views, and commands, can be parsed. Since we wish to add new commands, we need to extend such a module with the new declarations. For example, for the command `norm`, we can define the following `NORM-SIGN` module.

```
fmod NORM-SIGN is
  including FULL-MAUDE-SIGN .
  op norm_. : Bubble -> Command .
endfm
```

To simplify parsing, we usually do it in two steps. Note for example that in modules the parsing of equations requires considering the signature of the module being analyzed to be able to parse the terms. The pieces of text (any non-empty string of Maude identifiers) whose parsing is deferred for parsing in a second stage, are kept as *bubbles*. The idea is that for a language that allows modules with user-definable syntax—as it is the case for Maude—it is natural to see its syntax as a combined syntax, at two different levels: what we may call the *top level* syntax of the language, and the user-definable syntax introduced in each module (see [6] for details). The data type `Bubble` allows us to reflect this duality of levels in the syntax definition. With the operator `norm_.` we declare the syntax for the command, but keep its argument as a bubble. We shall discuss below how bubbles are handled.

In this signature, all command operators are defined of sort `Command`. We can also find in `FULL-MAUDE-SIGN` sorts `Module` and `View`, with the appropriate declarations. A supersort of these, `Input`, covers all valid inputs.

The metarepresentation of this module, which we shall use in calls to `metaParse` can be obtained as follows.

```
fmod META-NORM-SIGN is
  including META-FULL-MAUDE-SIGN .

  op NORM-GRAMMAR : -> FModule .
```

```
  eq NORM-GRAMMAR = addImports((including 'NORM-SIGN .), GRAMMAR) .
endfm
```

Here, GRAMMAR is the metarepresentation of the module FULL-MAUDE-SIGN, obtained in a similar way, which is defined in the module META-FULL-MAUDE-SIGN.

In the current version of Maude, input/output is accomplished by the predefined LOOP-MODE module, which provides a generic read-eval-print loop. In the case of Full Maude, the persistent state of the loop is given by a single object of class Database which maintains the database of the system. This object has an attribute db, to keep the actual database in which all the modules being entered are stored (a set of records), an attribute default, to keep the identifier of the current module by default, and attributes input and output to simplify the communication of the read-eval-print loop given by the LOOP-MODE module with the database. Using the notation for classes in object-oriented modules we can declare such a class as follows:

```
class DatabaseClass | db : Database,    default : ModName,
                      input : TermList, output : QidList .
```

The state of the read-eval-print loop is then given by an object of class DatabaseClass. In the case of Full Maude, the handling of the read-eval-print loop is defined in the modules DATABASE-HANDLING and FULL-MAUDE.

The module FULL-MAUDE includes the rules to initialize the loop (rule init), and to specify the communication between the loop—the input/output of the system—and the database (rules in and out). Depending on the kind of input that the database receives, its state will be changed, or some output will be generated. Since we want a new grammar to be used for parsing the inputs, we want to change the existing rules, that is, the FULL-MAUDE module must be redefined. We show here the relevant part of the new module. The handling of the nonvalid parses, and the declarations kept as in the original module are substituted by dots.

```
mod FULL-MAUDE is
 pr META-NORM-SIGN .          --- instead of META-SIGN
 pr NORM-DATABASE-HANDLING . --- instead of DATABASE-HANDLING

 var  O : Oid .                 var  X@Database : DatabaseClass .
 var  DB : Database .           var  MN : ModName .
 var  Atts : AttributeSet .     vars QIL QIL' : QidList .

 ...

 crl [in] :
  [QIL,
   < O : X@Database | db : DB, input : nilTermList,
     output : nil, default : MN, Atts >,
   QIL']
  => [nil,
      < O : X@Database | db : DB,
        input : getTerm(metaParse(NORM-GRAMMAR, QIL, 'Input)),
        output : nil,          ---   \
        default : MN, Atts >, ---    \
      QIL']                   ---     \
```

```
  if QIL =/= nil               ---      \
     /\ metaParse(NORM-GRAMMAR, QIL, 'Input) : ResultPair .
              ---           \             \
  ...         ---            \-------------\-- instead of GRAMMAR
end
```

This module includes NORM-DATABASE-HANDLING, which defines the behavior of the database upon new entries. NORM-DATABASE-HANDLING imports the module DATABASE-HANDLING, since we want to have the same behavior for already considered inputs, and add declarations for handling appropriately the additional command.

The key definition in NORM-DATABASE-HANDLING is the following rule.

```
rl [NORM] :
   < O : X@Database | db : DB, input : ('norm_.[T]),
      output : nil, default : MN, Atts >
   => < O : X@Database | db : DB, input : nilTermList,
         output : procCommand('norm_.[T], MN, DB),
         default : MN, Atts > .
```

This rule defines what to do when a norm command is entered. Note that this is the case if the result of the parsing in the rule in in the module FULL-MAUDE has the form 'norm_.[T], where T is a variable of sort Term representing a bubble. The function procCommand specifies what to do when such a command is entered, with MN and DB variables with values: the name of the current default module and the state of the database, respectively. In the case of the norm command, procCommand calls the function procNorm with the appropriate arguments, namely the name of the default module, the flatten module itself, the bubble representing the argument of the command, the variables in the default module, and the database. Note that depending on whether the default module is a built-in or not, and whether it is compiled or not, procCommand will do different things, so that the arguments for procNorm are obtained.

The function procNorm is in charge of evaluating the bubble given as argument of the norm command, calling the function metaNorm above, and then preparing the results (a list of quoted identifiers that will be passed to the output channel of the read-eval-print loop to be shown to the user).

```
op procNorm : ModExp Module Term OpDeclSet Database -> QidList .
```

Note that the norm and red commands have a similar syntax, that is, it may be of the form "norm <term> ." or "norm in <module> : <term> .". The analysis of such a bubble is accomplished by the function solveBubblesRed in Full Maude.

The complete specifications can be found in

http://www.dsic.upv.es/users/elp/toolsMaude

5 Extending Full Maude to handle the command `eval`

When considering how the evaluation command `norm` works for a concrete input term t, we understand that it is interesting to isolate the replacement restrictions needed to achieve the head-normal form of t (produced by the command `reduce`) from the restrictions needed to get its normal form (produced by the command `norm`). In the case of constructor normal forms, we can rather consider a constructor (prefix) context $C[\,]$ of the normal form of t such that $C[\,] \in \mathcal{T}(\mathcal{B} \cup \{\Box\}, \mathcal{X})$ for some $\mathcal{B} \subseteq \mathcal{C}$. The intuitive idea is that the set $\mathcal{B}$ considers those constructor symbols which are (or could be) present in the normal form of t. Then, reductions *below* the outermost defined symbols should be performed only up to (constructor) head-evaluation, in order to incrementally produce the normal form of t. In [2] a program transformation aimed at achieving this goal is given. The key idea for the transformation is to introduce a set $\mathcal{B}'_\tau$ of fresh constructor symbols having no replacement restrictions for each sort τ in the specification which is involved in producing the normal forms of interest. They are introduced as renamed versions $c' \in \mathcal{B}'_\tau$ of the original constructors $c \in \mathcal{B}_\tau$ such that $\mu(c') = \{1, \ldots, ar(c')\}$ (disregarding $\mu(c)$). Given a sort τ, the set $\mathcal{C}^*_\tau \subseteq \mathcal{C}$ is the set of constructor symbols that can be found in constructor terms of sort τ. For instance, $\mathcal{C}^*_{\mathtt{Nat}} = \{\mathtt{0}, \mathtt{s\_}\}$ and $\mathcal{C}^*_{\mathtt{List(Nat)}} = \{\mathtt{nil}, \mathtt{\_\_}, \mathtt{0}, \mathtt{s\_}\}$. Hence, the set $\mathcal{C}^*_\tau$ will tell us which constructor symbols must be renamed.

The renaming of the constructor symbols $c \in \mathcal{C}^*_\tau$ into new constructor symbols c' is performed by the rules:

$$\mathtt{quote}_{sort(c)}(c(x_1, \ldots, x_k)) \to c'(\mathtt{quote}_{sort(x_1)}(x_1), \ldots, \mathtt{quote}_{sort(x_k)}(x_k))$$

In practice, we use the overloading facilities of Maude and introduce a single (overloaded) symbol `quote`.

The evaluation of a term t would proceed by reducing $\mathtt{quote}(t)$ into a term with a constructor prefix context in $\mathcal{C}'$. Then, we perform a postprocessing that recovers the original names of the constructor symbols after the evaluation of the initial expression:

$$\mathtt{unquote}(c'(x_1, \ldots, x_k)) \to c(\mathtt{unquote}(x_1), \ldots, \mathtt{unquote}(x_k))$$

Again, the symbol `unquote` is conveniently overloaded.

The evaluation of quoted terms $\mathtt{quote}(t)$ is performed by introducing some additional rules to appropriately deal with the new symbols in the setting of the considered program $\mathcal{R}$. The transformation $\mathsf{V}^f(\mathcal{R})$ takes a defined symbol f as a parameter (usually the top symbol of t) and yields a new module which can be used to obtain the values of expressions t whose root symbol is f. By lack of space, we cannot give all details of the transformation and we refer the reader to [2] for a detailed explanation of its definition and conditions of use.

We have implemented a new command `eval` which uses this transformation to obtain the *value* (if any) associated to a given input expression t. In contrast to the command `norm`, we first transform the module and, then, we simply reduce the expression `unquote`(`quote`(t)) within the new module (using the `metaReduce` function). We define a function `transformCorrComp` which takes arguments of sort `Module` and `Qid` and returns the transformed module. Basically, `transformCorrComp` builds the set of rules and new symbols necessary for the transformation and adds them to the previous module; see [2] for details.

```
var  M : Module .          vars F Q Q' : Qid .
var  QIL : QidList .       var  IL : ImportList .
var  SS : SortSet .        var  SSDS : SubsortDeclSet .
var  OPDS : OpDeclSet .    var  MAS : MembAxSet .
var  EQS : EquationSet .   vars SortsF SortsForF : SortSet .

op transformCorrComp : Module Qid -> Module .

ceq transformCorrComp(M, F)
  = (fmod Q' is
       IL
       sorts SS .
       SSDS
       (OPDS
        genNewSignature(DF ConstructorsForSortF)
        genQuoteUnquote(SortsForF ; builtinSorts))
       MAS
       (EQS
        genEquationSetS(M, DF, DF ConstructorsForSortF)
        genEquationSetQuote(DF ConstructorsForSortF)
        genEquationSetUnquote(ConstructorsForSortF)
        genEquationSetID(builtinSorts))
     endfm)
  if (fmod Q is IL sorts SS . SSDS OPDS MAS EQS endfm) := M
  /\ Q' := qid("Q" + string(Q))
  /\ DF := definedTRS(M, F)
  /\ SortsF := getSortsOfQid(M, F)
  /\ SortsForF := cvSort(M, SortsF)
  /\ ConstructorsForSortF
       := filterConstructorSymbols(symbolsOfSorts(M, SortsForF)) .
```

As done for the command `norm`, we extend Full Maude so that the command `eval` can be entered by the user. We add the syntax for the new command `eval` defining the following `TRCOMPLETE-SIGN` module.

```
fmod TRCOMPLETE-SIGN is
  including FULL-MAUDE-SIGN .
  op eval_. : Bubble -> Command .
endfm
```

We also add the appropriate rules in a `TRCOMPLETE-DATABASE-HANDLING` module and redefine the module `FULL-MAUDE`, as for the `norm` command. The key definition in the module `TRCOMPLETE-DATABASE-HANDLING` is the following rule calling the function `procCommand`:

```
rl [EVAL] :
  < O : X@Database | db : DB, input : ('eval_.[T]),
     output : nil, default : MN, Atts >
  => < O : X@Database | db : DB, input : nilTermList, default : MN,
```

```
output : procCommand('eval_.[T], MN, DB), Atts > .
```

We have included some extra functions used in the sequence of calls from the function `procCommand` down to the function `transformCorrComp` presented above.

```
op procTRCorrComp : ModExp Module Term OpDeclSet Database
     -> QidList .
op preTransformCorrComp : Module Term Qid -> QidList .
```

The function `procCommand` calls the function `procTRCorrComp` with the appropriate arguments: the name of the default module, the flatten module itself, the bubble representing the argument of the command, the variables in the default module, and the database. The function `procTRCorrComp` is in charge of evaluating the bubble given as argument of the `eval` command, calling another extra function `preTransformCorrComp`, and then preparing the results (a list of quoted identifiers that will be passed to the output channel of the read-eval-print loop to be shown to the user). The function `preTransformCorrComp`, which finally calls `transformCorrComp`, performs two tests in order to determine whether the transformation is necessary or not. These two tests are associated to the following two facts:

(i) (cf. [2, Theorem 6.14]) If $\mathcal{R}$ is a left-linear and confluent Maude program and the strategy map φ of $\mathcal{R}$ is compatible with the canonical replacement map (i.e., $\mu^{\varphi} \in CM_{\mathcal{R}}$), then every ground constructor value of term $t = f(t_1, \ldots, t_k)$ can be computed by evaluating $\texttt{unquote}(\texttt{quote}(t))$ in the transformed program $\mathsf{V}^f(\mathcal{R})$.

(ii) (cf. [2, Theorem 5.4]) If $\mathcal{R}$ is a left-linear and confluent Maude program, the strategy map φ of $\mathcal{R}$ is compatible with the canonical replacement map, and $\varphi(c) = \{1, \ldots, ar(c)\}$ for every constructor symbol $c \in \mathcal{C}^*_\tau$, then every ground constructor value of a term t of sort τ can be computed with `red` in $\mathcal{R}$.

Thus, the first test checks whether the E-strategy map associated to the input module is not canonical and, in such case, the transformation is not applied. The second test checks whether the strategy maps associated to the constructor symbols included in $\mathcal{C}^*_\tau$ contain all the indices and, in such case, the transformation is not applied.

As an example of the use of the new command, let us consider again the computation of the value of the expression in Section 1:

```
Maude> (eval take(4, natsFrom(0)) .)
transforming module NAT-LIST for symbol take
transformed module QNAT-LIST is complete for defined symbols: take

reduce in QNAT-LIST :
  unquote(quote(take(4, natsFrom(0))))
result List`(Nat`) :
```

```
0 . 1 . 2 . 3 . nil
```

The (transformed) module internally used by the `metaReduce` function launched for the expression `unquote(quote(take(4, natsFrom(0))))` is given in Appendix A.

Furthermore, the reader should note that since the transformation is parametric with respect to a defined function symbol, the command `eval` is often more interesting than the command `norm`, since we can evaluate an expression within the transformed module for a concrete defined function symbol. This often improves the termination behavior of the evaluation process (see [2]). For this reason, we have implemented a variant of the command `eval_for_` allowing the explicit inclusion of the defined function symbol (which can be thought of as expressing the functionality we are interested in) to transform the module.

```
Maude> (eval natsFrom(0) for take .)
transforming module NAT-LIST for symbol take
transformed module QNAT-LIST is complete for defined symbols: take

reduce in QNAT-LIST :
  unquote(quote(natsFrom(0)))
result List‘(Nat‘) :
  0 . natsFrom(0 + 1)
```

Note that the expression `natsFrom(0)` enters in an infinite evaluation both using the command `eval` and `norm`:

```
Maude> (eval natsFrom(0) .)
Segmentation fault

Maude> (norm natsFrom(0) .)
Segmentation fault
```

Thus, with `eval_for_`, the non-termination of expression `natsFrom(0)` can be avoided while constructor normal forms for symbol `take` are provided. On the other hand, `eval` does not compute arbitrary normal forms but only constructor normal forms.

Finally, it is worth noting that built-in modules, sorts, and symbols included in Maude are preserved by adding the equations

$$\texttt{quote}(X) \to X \ \texttt{[owise]} \ .$$

$$\texttt{unquote}(X) \to X \ \texttt{[owise]} \ .$$

for built-in sorts such as `Bool`, `Int`, or `Nat`. Note that these equations have the attribute `owise` in order to not interfere with normal `quote` and `unquote` equations. This is done by the function `genEquationSetID`.

As for the `norm` command, the complete specification can be found in

http://www.dsic.upv.es/users/elp/toolsMaude

In this URL, one can also find declarations for making both commands avail-

able simultaneously.

6 Conclusions and future work

Maude is able to deal with infinite data structures and avoid infinite computations by using *strategy annotations* (see [17]). Maude strategy annotations are lists of non-negative integers associated to function symbols that specify the ordering in which the arguments are (eventually) evaluated in function calls. Some argument indices can eventually be missing from these lists thus improving the termination behavior of the program. However, they can eventually make the computation of the normal form(s) of some input expressions impossible.

We have used Full Maude to implement two new commands `norm` and `eval` which furnish Maude with the ability to compute (constructor) normal forms of initial expressions even when the use of strategy annotations together with the built-in computation strategy of Maude (i.e., command `red`) is not able to obtain them. The command `norm` performs a layered normalization of the initial expression until the normal form is reached (if any). On the other hand, `eval` uses a transformed program to obtain a different one which is able to obtain the constructor normal forms which correspond to a given input expression.

These commands have been integrated into Full Maude, making them available inside the programming environment like any other of its commands. The high level at which the specification/implementation of Full Maude is given makes this approach particularly attractive when compared to conventional implementations. The flexibility and extensibility that Full Maude affords has made the extension quite simple, and in a very short time.

Although both `norm` and `eval` are intended to achieve a common goal, they work quite differently. We have performed a simple comparison of the behavior of `norm` and `eval` regarding their efficiency. We have used the Full Maude modules in Section 1 together with a couple of new ones to implement a function

```
op factorsOf : Nat -> List(PNat) .
```

which produces the list of prime factors of a natural number (see the complete program in Appendix B) as follows:

```
Maude> (norm factorsOf(123456789) .)
reduce in FACTORS
   factorsOf(123456789)
result List‘(PNat‘) :
   (3 ^ 2).PNat .(3607 ^ 1).PNat .(3803 ^ 1).PNat
```

As expected, the function `factorsOf` tell us that $123456789 = 3^2 \cdot 3607 \cdot$

N	123456789	1234567890	12345654321	1234567654321	123456787654321
`norm`	2165331/3630	2165530/4230	1231/0	3145001/6990	5888/20
`eval`	2172567/4060	2167532/4500	2201/20	3147446/7340	6985/20
`eval`*	2166040/3510	2166074/3890	743/0	3145988/6370	5527/10

Fig. 1. Cost of computing `factorsOf(`N`)` in *rewriting steps/milliseconds*.

3803. The table in Figure 6 summarizes the cost of computing `factorsOf(`N`)` for five natural numbers N showing *rewriting steps/time (in milliseconds)*. The table shows that, in general, `norm` is usually more efficient than `eval`, but not so much. This is not surprising, though, since the use of `eval` involves computations using a transformed program which always adds some new rules. The row `eval`* corresponds to the direct use of the transformed program in Core Maude (without using Full Maude). It is interesting to note that the overloading which is introduced by the preprocessing step of `eval` (i.e., computing the transformed program) is not so high. This is more evident when we consider the factorization of the first, second and fourth numbers where the computational effort due to the factorization is much higher than the administrative overloading.

The new commands `norm` and `eval` permit to overcome the limits of strategy annotations regarding correctness and completeness of computations. An alternative way to address the problems introduced by the absence of some indices in the local strategies is including *negative* indices as part of strategy annotations to express that the corresponding arguments will be evaluated *on-demand*, where the 'demand' is an attempt to match an argument term with the left-hand side of a rewrite rule [22]. This often permits to recover correctness and completeness of computations while the program is still terminating. An operational model for on-demand strategy annotations, which is based on a suitable (and conservative) extension of the E-evaluation strategy of Maude (which only considers annotations given as natural numbers) has been proposed in [1]. In [8], we present an extension of Full Maude with the ability to accept and execute programs with on-demand strategy annotations according to the computational model proposed in [1].

References

[1] M. Alpuente, S. Escobar, B. Gramlich, and S. Lucas. Improving On-Demand Strategy Annotations. In M. Baaz and A. Voronkov, editors, *Proc. 9th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR'02*, Lecture Notes in Artificial Intelligence 2514:1-18, Springer, 2002.

[2] M. Alpuente, S. Escobar, and S. Lucas. Correct and complete (positive) strategy annotations

for OBJ. In F. Gadducci and U. Montanari, editors, *Proc. of the 4th International Workshop on Rewriting Logic and its Applications, WRLA'02*, *Electronic Notes in Theoretical Computer Science*, volume 71. Elsevier Sciences, 2004.

[3] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.-O. Stehr. Maude as a formal meta-tool. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99 - Formal Methods (Vol. II)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1684–1704. Springer-Verlag, 1999.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science* 285(2):187-243, 2002.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. The Maude 2.0 System. In R. Nieuwenhuis, editor, Proc. of *14th International Conference on Rewriting Techniques and Applications, RTA'03*, Lecture Notes in Computer Science 2706:76-87, Springer, 2003. Available at http://maude.cs.uiuc.edu.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 manual. Available in http://maude.cs.uiuc.edu., June 2003.

[7] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Lecture Notes. CSLI Publications, 2000.

[8] F. Durán, S. Escobar, and S. Lucas. On-demand evaluation for Maude. In S. Abdennadher and C. Ringeissen, editors, *Proceedings of 5th International Workshop on Rule-Based Programming (RULE'04)*, *Electronic Notes in Theoretical Computer Science*. Elsevier, 2004.

[9] F. Durán and J. Meseguer. An extensible module algebra for Maude. In C. Kirchner and H. Kirchner, editors, *Proceedings of 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*, pages 185–206. Elsevier, 1998. Available at http://www.elsevier.nl/locate/entcs/volume15.html.

[10] F. Durán. *A Reflective Module Algebra with Applications to the Maude Language*. PhD thesis, Universidad de Málaga, June 1999.

[11] S. Eker. Term Rewriting with Operator Evaluation Strategies. *Electronic Notes in Theoretical Computer Science*, volume 15, 20 pages, 1998.

[12] S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1-61, January 1998.

[13] S. Lucas. Termination of on-demand rewriting and termination of OBJ programs. In *Proc. of 3rd International Conference on Principles and Practice of Declarative Programming, PPDP'01*, pages 82-93, ACM Press, 2001.

[14] S. Lucas. Termination of Rewriting With Strategy Annotations. In R. Nieuwenhuis and A. Voronkov, editors, *Proc. of 8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR'01*, Lecture Notes in Artificial Intelligence 2250:669-684, Springer, 2001.

[15] S. Lucas. Context-sensitive rewriting strategies. *Information and Computation*, 178(1):293-343, 2002.

[16] S. Lucas. Termination of (Canonical) Context-Sensitive Rewriting. In S. Tison, editor, *Proc. 13th International Conference on Rewriting Techniques and Applications, RTA'02*, Lecture Notes in Computer Science 2378:296-310, Springer, 2002.

[17] S. Lucas. Semantics of programs with strategy annotations. Technical Report DSIC-II/08/03, DSIC, Universidad Politécnica de Valencia, 2003.

[18] S. Lucas. Termination of programs with strategy annotations. Technical Report DSIC II/20/03, Universidad Politécnica de Valencia, Sep. 2003.

[19] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic*. Springer-Verlag, 1999.

[20] N. Martí-Oliet and J. Meseguer. Rewriting logic as a logical and semantic framework. In D. M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Second Edition, Volume 9*, pages 1–87. Kluwer Academic Publishers, 2002.

[21] T. Nagaya. Reduction Strategies for Term Rewriting Systems. PhD Thesis, School of Information Science, Japan Advanced Institute of Science and Technology, March 1999.

[22] M. Nakamura and K. Ogata. The evaluation strategy for head normal form with and without on-demand flags. In K. Futatsugi, editor, *Proc. of 3rd International Workshop on Rewriting Logic and its Applications, WRLA'00*, *Electronic Notes in Theoretical Computer Science*, volume 36, 17 pages, 2001.

A Module resulting from the transformation in the command `eval` for the running example

```
fmod QNAT-LIST is
  protecting NAT .
  protecting BOOL .
  protecting INT .
  sorts List‘(Nat‘) .
  subsort Nat < List‘(Nat‘) .
  op _._ : Nat List‘(Nat‘) -> List‘(Nat‘) [strat(1 0)ctor] .
  op _._’ : Nat List‘(Nat‘) -> List‘(Nat‘) [ctor] .
  op nil : -> List‘(Nat‘) [ctor] .
  op nil’ : -> List‘(Nat‘) [ctor] .
  op quote : List‘(Nat‘) -> List‘(Nat‘) [strat(0)] .
  op quote : Nat -> Nat [strat(0)] .
  op unquote : List‘(Nat‘) -> List‘(Nat‘) [strat(1 0)] .
  op unquote : Nat -> Nat [strat(1 0)] .
  op unquoteF_._ : Nat List‘(Nat‘) -> List‘(Nat‘) [ctor] .
  op natsFrom : Nat -> List‘(Nat‘) .
  op take : Int List‘(Nat‘) -> List‘(Nat‘) .
  op take’ : Int List‘(Nat‘) -> List‘(Nat‘) .
  eq natsFrom(N:Nat) = N:Nat . natsFrom(N:Nat + 1) .
  eq quote(nil) = nil’ .
  eq quote(V0:Nat . V1:List‘(Nat‘))
    = quote(V0:Nat). quote(V1:List‘(Nat‘))’ .
  eq quote(take(V0:Int,V1:List‘(Nat‘)))
    = take’(V0:Int,V1:List‘(Nat‘)) .
  eq take(0,Z:List‘(Nat‘)) = nil .
  eq take(N:Int,X:Nat . Z:List‘(Nat‘))
    = X:Nat . take(N:Int - 1,Z:List‘(Nat‘)) .
  eq take’(0,Z:List‘(Nat‘)) = nil’ .
  eq take’(N:Int,X:Nat . Z:List‘(Nat‘))
    = quote(X:Nat). take’(N:Int - 1,Z:List‘(Nat‘))’ .
  eq unquote(nil’) = nil .
  eq unquote(V0:Nat . V1:List‘(Nat‘) ’)
    = unquoteF unquote(V0:Nat). unquote(V1:List‘(Nat‘)) .
  eq unquoteF V0:Nat . V1:List‘(Nat‘) = V0:Nat . V1:List‘(Nat‘) .
  eq quote(X:List‘(Nat‘)) = X:List‘(Nat‘) [owise] .
  eq quote(X:Nat) = X:Nat [owise] .
  eq unquote(X:List‘(Nat‘)) = X:List‘(Nat‘) [owise] .
  eq unquote(X:Nat) = X:Nat [owise] .
endfm
```

B Example producing the list of prime factors of a natural number in Full Maude

```
(fmod LAZY-LIST(X :: TRIV) is
  protecting INT .
  sort List(X) .
  subsort X@Elt < List(X) .
  op nil : -> List(X) [ctor] .
```

```
  op _._ : X@Elt List(X) -> List(X) [ctor strat (1 0)] .
  op take : Int List(X) -> List(X) .
  var N : Int .
  var X : X@Elt .
  var Z : List(X) .
  eq take(0, Z) = nil .
  eq take(N, X . Z) = X . take(N - 1, Z) .
 endfm)

(view Nat from TRIV to NAT is
  sort Elt to Nat .
 endv)

(fmod NAT-LIST is
  protecting LAZY-LIST(Nat) .
  op natsFrom : Nat -> List(Nat) .
  var N : Nat .
  eq natsFrom(N) = N . natsFrom(N + 1) .
 endfm)

(fmod PRIMES is
  pr INT .
  pr NAT-LIST .
  op filter : List(Nat) Int Int -> List(Nat) .
  op sieve : List(Nat) -> List(Nat) .
  op primes : -> List(Nat) .
  vars X M N : Int .
  vars Y Z : List(Nat) .
  eq filter(X . Y, 0, M) = 0 . filter(Y, M, M) .
  eq filter(X . Y, N, M) = X . filter(Y, N - 1, M) .
  eq sieve(0 . Y) = sieve(Y) .
  eq sieve(N . Y) = N . sieve(filter(Y, N - 1, N - 1)) .
  eq primes = sieve(natsFrom(2)) .
 endfm)

(fmod PNAT is
  pr NAT .
  sort PNat .
  op _^_ : Nat Nat -> PNat [ctor] .
 endfm)

(view PNat from TRIV to PNAT is
  sort Elt to PNat .
 endv)

(fmod FACTORS is
  pr PRIMES .
  pr LAZY-LIST(Nat) .
  pr LAZY-LIST(PNat) .
  op factors : Nat List(Nat) Nat -> List(PNat) .
  op factorsOf : Nat -> List(PNat) .
  vars N P E : Nat .
  var  PS : List(Nat) .
```

```
 eq factors(1, P . PS, E) = P ^ E .
 eq factors(N, P . PS, 0)
   = if P divides N
     then factors(N quo P, P .  PS, 1)
     else factors(N, PS, 0)
     fi .
 ceq factors(N, P . PS, E)
   = if P divides N
     then factors(N quo P, P . PS, E + 1)
     else (P ^ E) . factors(N, PS, 0)
     fi
   if N > 1 /\ E > 0 .
 eq factorsOf(1) = 1 ^ 1 .
 ceq factorsOf(N) = factors(N, primes, 0) if N > 1 .
endfm)
```